UNCLASSIFIED

UNLIMITED DISTRIBUTION

Rapidly Evolving Distributed Systems by Bridging the Deployment Gap

Final Report

September 28, 2000

Alexander Wolf
(Principal Investigator)

Dennis Heimbigner
(Co-Principal Investigator)

Computer Science Department
University of Colorado
Boulder, CO 80309-0430
Email: alw@cs.colorado.edu
Telephone: 303-492-5263
Fax: 303-492-2844

Final Report for

DARPA Order E575
AFRL Contract F30602-98-2-0163

Contract Period: 4/6/98–6/28/2000
Contract Amount: $300,000

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 28, 2000	Final Report for period 4/6/98-6/28/2000

4. TITLE AND SUBTITLE
Rapidly Evolving Distributed Systems by Bridging the Deployment Gap

5. FUNDING NUMBERS
DARPA Order E575
AFRL Contract F30602-98-2-0163

6. AUTHOR(S)
Alexander Wolf, Associate Professor, University of Colorado
Dennis Heimbigner, Research Associate Professor, University of Colorado

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Regents of the University of Colorado
Office of Contracts and Grants
Boulder, CO 80309-0572

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA/ISO, 3701 N. Fairfax Dr., Arlington. VA 22203-1704
AFRL/IFTD, 525 Brooks Rd., Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U. S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution A: Approved for public release, distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
The University of Colorado EDCS project has addressed problems in managing the configurations of evolved systems and deploying those systems back out into the field. The essential premise of this project was that configuration and deployment of distributed systems of systems is a critical piece of the cycle of evolutionary development of complex software systems. The University of Colorado EDCS project has been successful in achieving its objective: producing innovative, useful, and interesting research results in the areas of software configuration and deployment. These research results were embodied in five prototype systems targeting five configuration and deployment problems: NUCM (distributed CM), SRM (software release), DVS (distributed development), Software Dock (distributed wide-area deployment), and {\Siena} (internet-scale event notification). The results from this project have been widely disseminated in the form of publications, software distributions to over 600 sites, technical transfers to commercial practice, and through the graduation of quality Ph.D. and M.S. students.

14. SUBJECT TERMS
Configuration management, deployment, event notification, software engineering, distributed computing

15. NUMBER OF PAGES
38

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Abstract

The University of Colorado EDCS project has addressed problems in managing the configurations of evolved systems and deploying those systems back out into the field. The essential premise of this project was that configuration and deployment of distributed systems of systems is a critical piece of the cycle of evolutionary development of complex software systems. The University of Colorado EDCS project has been successful in achieving its objective: producing innovative, useful, and interesting research results in the areas of software configuration and deployment. These research results were embodied in five prototype systems targeting five configuration and deployment problems: NUCM (distributed CM), SRM (software release), DVS (distributed development), Software Dock (distributed wide-area deployment), and Siena (internet-scale event notification). The results from this project have been widely disseminated in the form of publications, software distributions to over 600 sites, technical transfers to commercial practice, and through the graduation of quality Ph. D. and M. S. students.

Contents

List of Figures

List of Tables

1 Introduction

Configuration and deployment of distributed systems of systems is an essential piece of the evolutionary development of complex software systems (EDCS). Software Evolution is associated with a cyclic development, implementation, and deployment process that starts with recognition that an existing software system is failing to meet its requirements or has had new requirements levied against its operation. A software redevelopment process is performed to modify its design and to implement a new version of a system capable of meeting its revised requirements. After redevelopment is complete, it is necessary to take the crucial step of deploying the evolved software back into the field to "complete the evolutionary cycle."

This project, referred to as University of Colorado EDCS, targeted the last step of the process and addressed problems in managing the configurations of evolved systems and deploying those systems back out into the field. This EDCS project was intended to be closely tied to the Arcadia project (contract F30602-94-C-0253), and was to provide a path by which configuration management and deployment research from the Arcadia project could be inserted into the DARPA EDCS program.

2 Project Objectives and Approach

Wide-area networks have become an essential context for many Department of Defense software systems. Currently, DOD operates over 100 wide-area networks, and this number will increase as a result of new programs such as Battlefield Awareness (BADD), Command Post of the Future (CPOF), Global Information Grid (GIG), and Joint Battlespace Infosphere (JBI). The stated goal underlying this trend is to enable the movement of information at all levels, replacing the movement of people with the movement of information.

Inherent in the existence of these global networks is an opportunity to leverage the connectivity of the network for software system configuration and deployment. The advantages of network deployment include the following.

- *Timeliness* — As soon as a new software system version or update becomes available, users can be given access to it.

- *Continuous evolution* — The semi-continuous connectivity offered by a network allows software producers to offer a much higher level of service to software consumers, moving beyond mere installation to encompass other activities such as activation, update, and adaptation. The resulting benefit is a lower total cost of ownership because less effort must be expended on maintaining deployed software.

- *Reuse* — The systems developed by software producers are more visible and more easily incorporated into larger systems, thus enhancing the reuse of a given system and promoting the development of systems of systems.

- *Recovery/Repair* — Network based distribution provides a repository of components that can be used as a baseline for detecting corrupted systems and as a source of components for purposes of repair.

In addition, there is an opportunity to support complex systems of systems — systems composed from component subsystems — where the components come from multiple sources and where there are many relationships among the components that must be honored and maintained.

The overall approach of this project was to develop new methods, techniques, and approaches for providing for the configuration management and deployment of complex software systems into a distributed environment.

Section 3 provides a brief description of the research produced under this contract. In all cases, the corresponding research publications should be referenced to obtain the details.

3 Results

The primary accomplishment of this project was the development of new approaches for supporting the configuration and deployment of complex software systems. We are at the forefront in configuration management research, and we have brought our capabilities in this area to the DARPA EDCS program.

The detailed accomplishments of this project fall into four categories: software prototypes, technical transfer, Ph. D. and M. S. students graduated, and publications. The first three are detailed in the following sections. A reverse chronological list of all publications is provided in Section 5.

3.1 Software Prototypes

The main vehicle for our research has been the development of a number of research prototype software systems. Each of these systems embodies important new capabilities in the area of configuration management and deployment. Five prototypes were developed in whole or part under this project.

1. *NUCM* – a generic, tailorable, peer-to-peer repository supporting distributed Configuration Management.

2. *SRM* – a tool to manage the release of multiple, interdependent software systems from distributed sites.

3. *DVS* – a tool to support distributed authoring and versioning of documents with complex structure, and to support multiple developers at multiple sites over a wide-area network.

4. *Software Dock* - a distributed, agent-based framework supporting software system deployment over a wide-area network.

5. *Siena* - an Internet-scale distributed event notification service allowing applications and people to coordinate in such activities as updating software system deployments.

The objectives, approach, and contributions of these prototypes are described in the following sections.

3.1.1 NUCM

NUCM [21, 26] is a generic, peer-to-peer repository supporting distributed Configuration Management (CM). Its programmatic interface allows for the rapid construction and evolution of CM systems, whereas its underlying distribution mechanism facilitates Configuration Management in the context of large-scale, wide-area software development.

NUCM separates CM repositories, which are the stores for versions of software artifacts and information about these artifacts, from CM policies, which are the specific procedures for creating, evolving, and assembling versions of artifacts maintained in the repository. Combined, a CM repository and a CM policy comprise a complete CM system. But it is their separation into two architectural components that, through reuse of the NUCM CM repository, facilitates the rapid development of complete CM systems.

With NUCM's generic programmatic interface it becomes feasible to develop a CM system that specifically supports and is tailored to an organization's internal software development process and policies. Until now, an organization was forced to buy a commercial CM system and adopt the process and policies incorporated in the acquired CM system. NUCM reverses this approach and instead allows the CM system to be specialized to the actual process and policies taking place.

NUCM provides the following benefits to a CM system developer:

- Rapid development. NUCM's reusable CM repository, combined with its generic interface, allows for the rapid construction of complete CM systems.
- Distributed operation. Any CM system developed with NUCM inherits NUCM's distributed nature, and can have CM clients and servers spread across the world.
- Scalability. NUCM's peer-to-peer architecture, combined with its lightweight implementation, presents a CM system developer with a scalable repository capable of operating in wide-area, large-scale Inter- and Intranets.
- Flexibility. The NUCM programmatic interface is generic, and supports the creation of a wide variety of CM policies.
- Type independence. NUCM can store and version any type of artifact.
- Evolvability. The NUCM repository supports the controlled evolution of artifacts through its versioning interface.

Data Model. The data model of NUCM is based on a flexible grouping mechanism in which atoms (individually versioned artifacts) and collections (groups of versioned artifacts) are treated identically. The data model maps naturally into the file system so that existing tools can manipulate the artifacts in their native environment. Furthermore, it is policy independent, and does not imply any relationship among the versions of an artifact.

The NUCM data model is analogous to that of a distributed, versioned file system with links and attributes. NUCM models artifacts as files and collections of artifacts as directories. Similar to a file system, collections (directories) can contain both artifacts (files) and other collections. Again, similar to a file system, NUCM supports links between collections and artifacts, so that the same artifact can be referenced in any number of collections. Figure 1 illustrates an example of the data model.

The NUCM versioning schema is orthogonal to the data model. In NUCM, artifacts as well as collections can have versions. The versioning schema is also completely independent of the relationships occurring between artifacts and collections. Two different versions of a collection can contain different versions of the same artifacts and/or completely different artifacts.

Distribution Model. NUCM provides the concepts of physical and logical repositories. A physical repository is the actual store for some set of artifacts at a particular site. A logical repository is a group of one or more repositories acting as a single repository. CM policies interact with a logical repository and can therefore manipulate any of the artifacts irrespective of physical location. Many different distribution topologies can be modeled by NUCM, such as client-server or peer-to-peer. NUCM physical repositories and CM policies can be distributed throughout the world, while all are part of a single CM system.

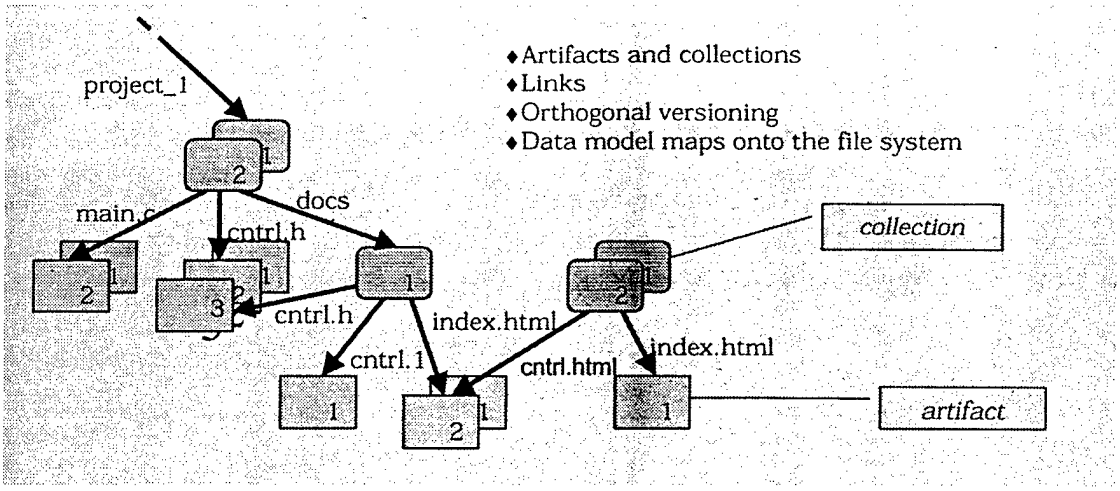

Figure 1: NUCM Data Model Example.

Generic Programmatic Interface. NUCM's programmatic interface supports CM system developers with a policy programming language. For example, the familiar check-in/check-out policy reduces to:

- check-out: open + testandsetattribute + initiatechange
- check-in: commitchange + removeattribute

This simplicity is intrinsic to NUCM; its interface functions have been carefully tuned to be simple yet powerful.

Experience. NUCM is in use in two systems that are publicly available, SRM and DVS, as well as one experimental system, WebDAV. The discussion of SRM is in Section 3.1.2 and the discussion of DVS is in Section 3.1.3. Our interest in WebDAV (Web Distributed Authoring and Versioning) stems from the participation of one of our members, André van der Hoek, in the initial standardization working group. This also led us to construct the first implementation of WebDAV. This was possible only because of the existence of NUCM, which made the effort to produce a WebDAV server relatively easy.

Figure 2 shows the interface for our WebDAV server operating through a NetScape browser. The important capability provided by WebDAV is that it allows one to edit web pages. Our prototype is actually more capable than the final WebDAV because it supports version trees over web pages. The graph show in that figure illustrates the version tree and can be used to retrieve specific versions. Our prototype was based on a near final draft WebDAV standard. The final standard removed versioning and deferred its inclusion to a later time.

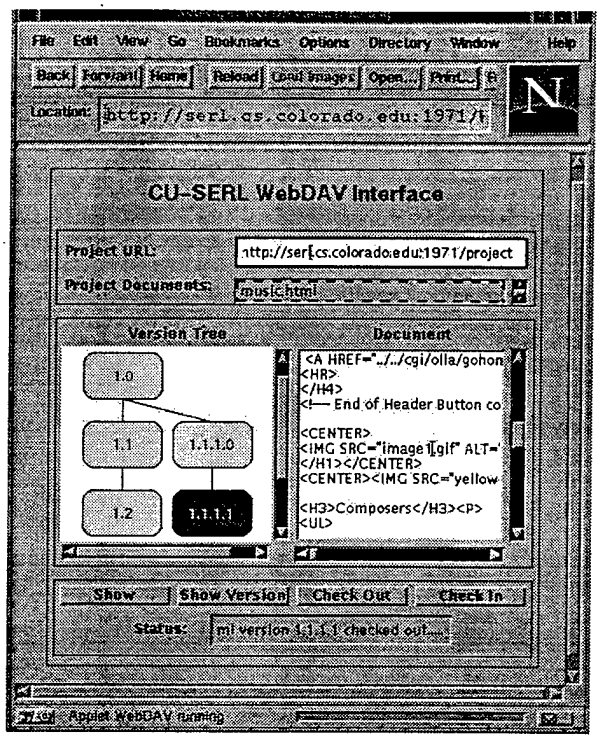

Figure 2: NUCM WebDAV Browser Interface.

Both the development time and development effort of these systems (WebDAV, SRM, and DVS) were greatly reduced due to the use of NUCM. For example, DVS is a fully functional, distributed versioning system that required only 1500 new lines of C source code.

3.1.2 SRM

Software release management is the process through which software is made available to and obtained by its users. Complicating software release management is the increasing tendency for software to be constructed as a "system of systems", assembled from pre-existing, independently produced, and independently released systems. In these situations, accurately managing dependencies among the systems is critical to the successful deployment of the system of systems.

SRM is a tool that addresses the software release management challenge. It supports the release of systems of systems from multiple, geographically distributed organizations. In particular, SRM tracks dependency information to automate and optimize the retrieval of components. Both developers and users of software systems are supported by SRM. Developers are supported by a simple release process that hides distribution. Users are supported by a simple retrieval process that allows the retrieval, via the Web, of a system of systems in a single step and as a single package.

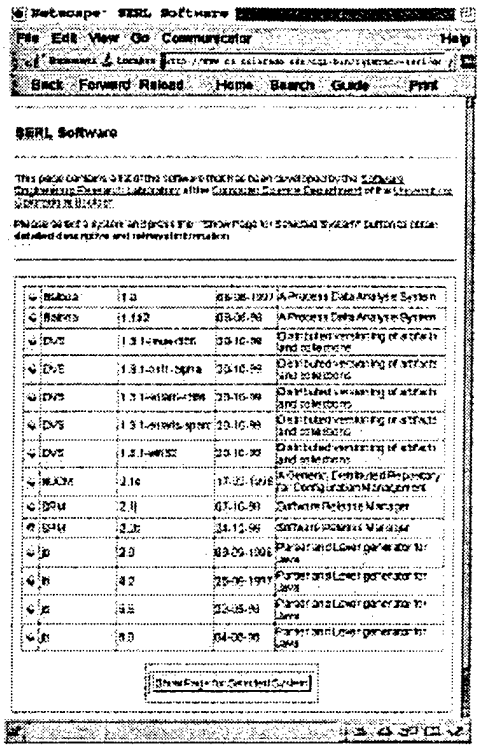

Figure 3: SRM Client Download Interface (using Netcape).

SRM provides the following benefits to an organization:

- *Process automation.* SRM incorporates and enforces a standard and fully automated release process.
- *Consistency.* Users always receive a consistent system of systems.
- *Flexibility.* SRM supports multiple release tracks, each with its own set of users that have access to its releases.
- *Scalability.* SRM can be configured to support an arbitrary number of cooperating organizations.
- *Web integration.* Software that is available on the Web from "non-SRM" organizations is integrated by SRM.
- *Uniformity.* All releases from all organizations can be released via the same mechanism.
- *Evolvability.* SRM's flexible dependency mechanism supports the evolution of software through multiple versions, each potentially with different dependencies.

Because of its versatility, SRM serves many different settings:

- An organization uses SRM as its release mechanism of choice to publish software on the Web. Various release groups are set up to distinguish alpha, beta, and production releases.

- An organization uses SRM as an intermediary between CM systems. For example, one department might use PCMS, whereas another one uses RCS. SRM can be used to ship and track updates that are sent back and forth.

- A group of loosely coupled organizations uses SRM as its unifying release mechanism of choice. A single server is placed at one of the organizations to which all other organizations release their software using an SRM client.

- A group of tightly cooperating, geographically dispersed organizations uses SRM to release their software both to the other organizations as well as to the outside world. Each organization maintains its own SRM server, but all SRM servers cooperate to present users with a single view.

Prototype. The SRM prototype presents two interfaces to the world. Figure 3 is the interface presented to users wishing to obtain software from an SRM repository. The interface is actually a web page produced by SRM as a *cgi-bin* script. This initial page shows all of the systems available for download through SRM. The user checks a system and pushes the button at the bottom.

SRM provides a second web page (Figure 4) that provides information about the chosen system. The bottom of this page (Figure 5), shows other systems upon which the chosen system depends. The set of dependencies is shown both graphically and in a list of user choosable items. The user is offered the option of:

- Obtaining the system alone,

- Obtaining the system with all of the systems upon which it depends, or

- Obtaining the system with a selected set of the systems upon which it depends.

SRM presents a different set of interfaces for users wishing to insert software into the SRM repository. Examples of this SRM user interface are illustrated in Figure 6 shows the initial interface. Users are provided a menu of options.

Figure 7 shows the interface that results when the modify option is chosen and SRM version 2.2b is chosen. Users are expected to fill in this form for an initial upload, and modify it otherwise. This page describes the software and indicates how the SRM repository is to obtain the software (typically as a tar file).

As part of the software release process, a user is prompted to indicate the other software systems upon which their system is dependent. Figure 8 shows this interface. A client selects

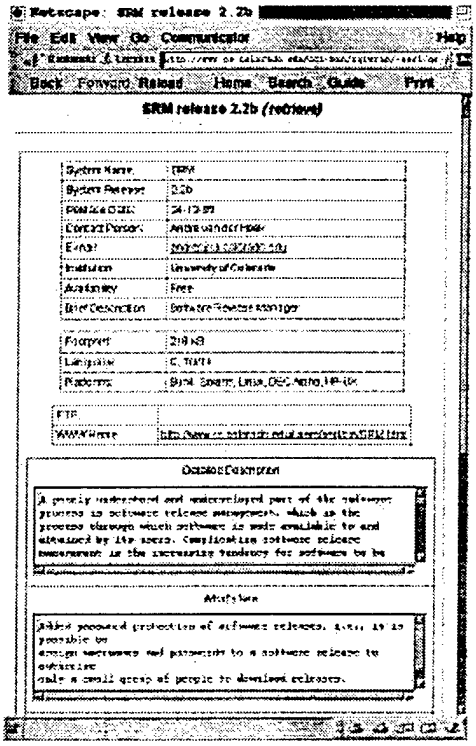

Figure 4: SRM Download Information Interface.

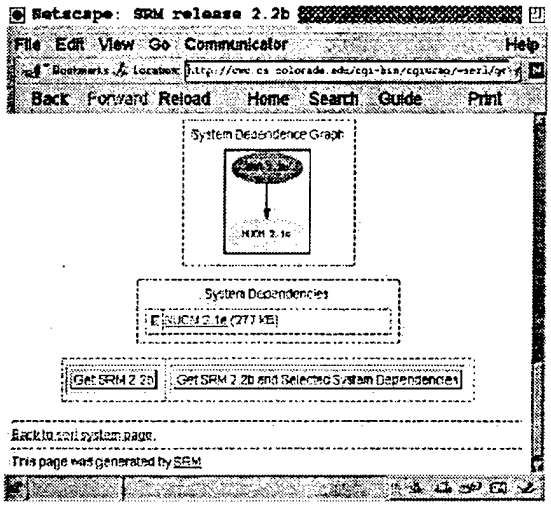

Figure 5: SRM Download Dependencies Interface.

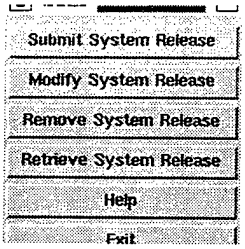

Figure 6: SRM Upload Menu.

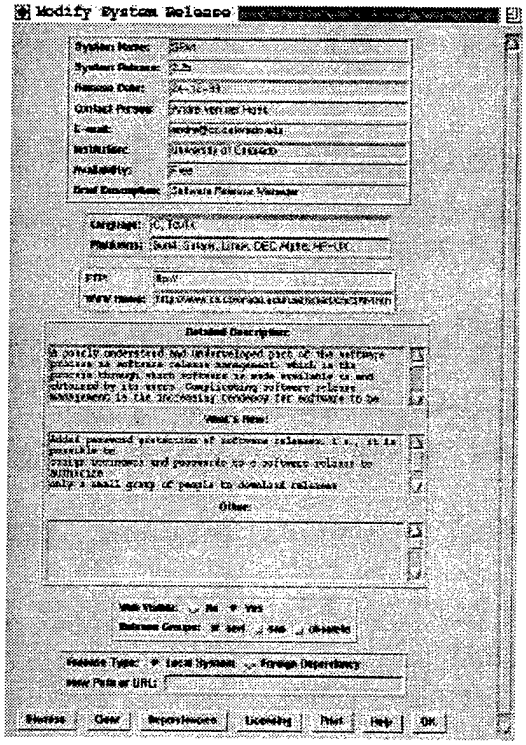

Figure 7: SRM Upload Interface.

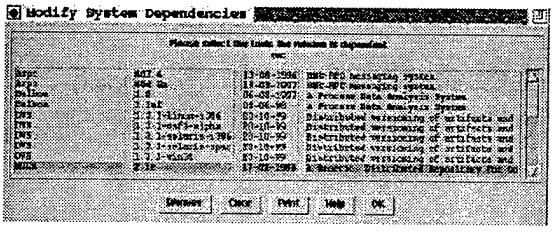

Figure 8: SRM Upload Dependency Selection Interface.

the list of supporting systems within SRM. This allows a retrieving client to obtain everything needed in one package. Finally, a user can specify a license to be provided to the user at download time (not shown).

Experience. SRM currently serves as the release mechanism for the software developed by the University of Colorado SERL group(*http://www.cs.colorado.edu/serl/software*) and by the University of Massachusetts LASER group (*http://laser.cs.umass.edu/tools/*). During the lifetime of the EDCS program, SRM was also used as the primary release mechanism for software produced by the EDCS projects. A central server, located at the Software Engineering Institute, served as the repository to which participating organizations released their systems. Subsequently, these systems were retrieved by users from all over the world.

3.1.3 DVS

DVS is a revision control system supporting distributed authoring and versioning of documents with complex structure. It supports multiple developers at multiple sites over the Internet. DVS differs from most other systems in allowing each document to be located at a different site, but shared and modified by users at all sites.

DVS is implemented on top of NUCM (Section 3.1.1), which allows DVS to be very light-weight. This is in contrast to existing commercial systems that have similar properties, but which are costly and bulky to install. In particular, DVS's physical repositories (below) are realized by NUCM servers, while the NUCM library provides basic access to artifacts, workspace management, and distribution.

The architecture of DVS (Figure 9) is composed of one logical repository and one or more workspaces. The logical repository contains artifacts that are under configuration management. Internally, the logical repository is realized by one or more physical repositories. A workspace is a per-user environment in which artifacts can be viewed, copied, and changed. DVS regulates the interactions between a workspace and the logical repository, for example, by checking in and out artifacts.

The data model implemented by DVS is an extension of the underlying NUCM model (see Section 3.1.1 and Figure 1). It provides a distributed, versioned file system with links and attributes. DVS models artifacts as files and collections of artifacts as directories. Similar to a file system, collections (directories) can contain both artifacts (files) and other collections. Again, similar to a file system, DVS supports links between collections and artifacts, so that the same artifact can be referenced in any number of collections.

NUCM itself specifies no specific versioning policy. So a major part of DVS is concerned with the definition and implementation of such a specific versioning policy. In this case, DVS implements simple linear versioning with versions numbered 1, 2, etc. The DVS versioning schema is orthogonal to the data model. In DVS, artifacts as well as collections can have versions. The versioning schema is also completely independent of the relationships occurring

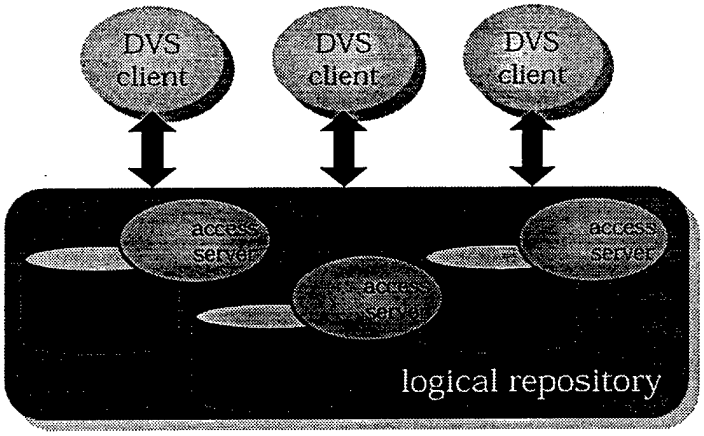

Figure 9: DVS Architecture.

between artifacts and collections. Two different versions of a collection can contain different versions of the same artifacts and/or completely different artifacts.

The mapping between the logical repository and the physical storage can be arbitrarily customized at the level of granularity of the single artifact. In other words, every artifact can be stored in a different repository, allowing the author to exploit "locality" by storing each artifact closer to the main author or the person that will access it most frequently.

Prototype. DVS consists of thirteen basic commands: *co*, *ci*, *close*, *link*, *unlink*, *lock*, *unlock*, *list*, *log*, *setlog*, *printlocks*, *whatsnew*, *sync*.

Most DVS commands can operate recursively following either the structure of the workspace or the structure of collections in the repository. The command *co* and *ci* respectively check out and check in versioned entities. Both *co* and *ci* can be applied to artifacts and collections. The *co* command can optionally lock a file and open it for change provided no one else holds the lock. The *ci* command requires that the file be currently checked out for change. Locks on artifacts can be directly acquired or released with *lock* and *unlock*. When inserting a new artifact with *ci*, an implicit link is also created with the current working collection. The *link* and *unlink* commands explicitly create and remove links between artifacts and collections.

When storing and retrieving artifacts to and from the repository, DVS records some meta-data together with each artifact or collection. Typically, a version log is maintained for each artifact. *log*, *setlog*, *printlocks*, and *list* are used to access those meta-data.

Besides the basic access and data model manipulation functions, DVS provides a set of utility services that facilitate distributed cooperation. They are *whatsnew* and *sync*. The *whatsnew* command informs a user of new revisions of artifacts and the *sync* command brings the content of the workspace up to date with respect to the content of the repository.

Experience. Originally, the purpose of DVS was to validate the NUCM approach, but now it is in regular use by SERL for distributed document development. DVS has also been used in several authoring efforts involving people from up to five sites distributed across the United States.

3.1.4 Software Dock

The connectivity of large networks, such as the Internet, is affecting how software deployment is being performed. The simple notion of providing a complete installation procedure for a software system on a CD-ROM is giving way to a more sophisticated notion of ongoing cooperation and negotiation among software producers and consumers. This connectivity and cooperation allows software producers to offer their customers high-level deployment services that were previously not possible. In the past, only software system installation was widely supported, but already support for the update process is becoming more common. Support for other software deployment processes, though, is still virtually non-existent.

New software deployment technologies are necessary if software producers are expected to accept more responsibility for the long-term operation of their software systems. In order to support software deployment, new deployment technologies must:

- operate on a variety of platforms and network environments, ranging from single sites to the entire Internet,
- provide a semantic model for describing a wide range of software systems in order to facilitate some level of software deployment process automation,
- provide a semantic model of target sites for deployment in order to describe the context in which deployment processes occur, and
- provide decentralized control for both software producers and consumers.

The *Software Dock* [11, 13, 14, 15, 22, 27] research project addresses many of these concerns. The Software Dock is a system of loosely coupled, cooperating, distributed components. The Software Dock supports software producers by providing the *release dock* that acts as a repository of software system releases. At the heart of the release dock is a standard semantic schema for describing the deployment requirements of software systems. The *field dock* component of the Software Dock supports the consumer by providing an interface to the consumer's resources, configuration, and deployed software systems. The Software Dock employs agents that travel from release docks to field docks in order to perform specific software deployment tasks while docked at a field dock. The agents perform their tasks by interpreting the semantic descriptions of both the software systems and the target consumer site. A wide-area event system connects release docks to field docks and enables asynchronous, bi-directional connectivity.

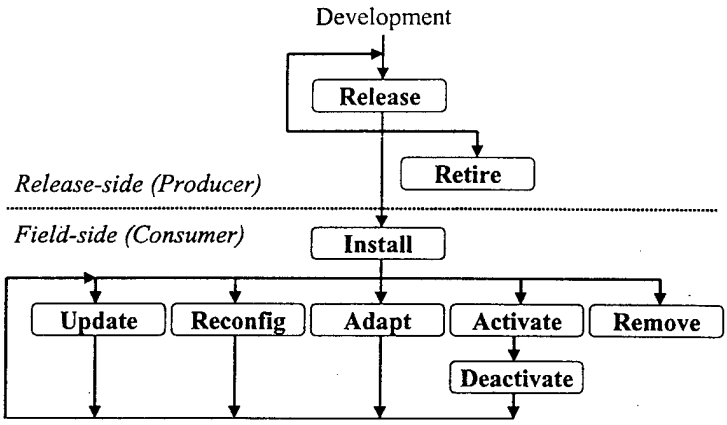

Figure 10: Deployment Life Cycle.

Software Deployment Life Cycle. In the past, software deployment was largely defined as the installation of a software system; a view of software deployment that is simplistic and incomplete. Software deployment is actually a collection of interrelated activities that form the software deployment life cycle. This life cycle, as defined by this research and diagramed in Figure 10, is an evolving collection of processes that include release, retire, install, activate, deactivate, reconfigure, update, adapt, and remove. Defining this life cycle is important because it indicates the new kinds of activities that the software producer may want to provide when moving beyond the mere installation of software. The resulting benefit to the software consumer is a lowered total cost of ownership since less effort is required to maintain the software that they own.

Architecture. The Software Dock research project addresses support for software deployment processes by creating a framework that enables cooperation among software producers themselves and between software producers and software consumers. The Software Dock architecture (Figure 11) defines components that represent these two main participants in the software deployment problem space. The release dock represents the software producer and the field dock represents the software consumer. In addition to these components the Software Dock employs agents to perform specific deployment process functionality and a wide-area event system to provide connectivity between the release docks and the field docks.

In the Software Dock architecture, the release dock is a server residing within a software producing organization. The purpose of the release dock is to serve as a release repository for the software systems that the software producer provides. The release dock provides a Web-based release mechanism that is not wholly unlike the release mechanisms that are currently in use; it provides a browser-accessible means for software consumers to browse and select software for deployment.

The release dock, though, is more sophisticated than most current release mechanisms. Within the release dock, each software release is described using a standard deployment

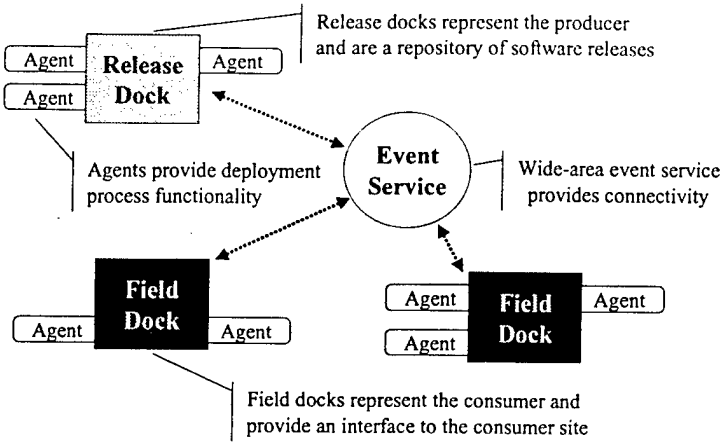

Figure 11: Software Dock Architecture.

schema; the details of standard schema description for software systems are presented in Section 4. Each software release is accompanied with generic agents that perform software deployment processes by interpreting the description of the software release. The release dock provides a programmatic interface for agents to access its services and content. Finally, the release dock generates events as changes are made to the software releases that it manages. Agents associated with deployed software systems can subscribe for these events to receive notifications about specific release-side occurrences, such as the release of an update.

The field dock is a server residing at a software consumer site. The purpose of the field dock is to serve as an interface to the consumer site. This interface provides information about the state of the consumer site's resources and configuration; this information provides the context into which software systems from a release dock are deployed. Agents that accompany software releases "dock" themselves at the target consumer site's field dock. The interface provided by the field dock is the only interface available to an agent at the underlying consumer site. This interface includes capabilities to query and examine the resources and configuration of the consumer site; examples of each might include installed software systems and the operating system configuration.

The release dock and the field dock are very similar components. Each is a server where agents can "dock" and perform activities. Each manages a standardized, hierarchical *registry* of information that records the configuration or the contents of its respective sites and creates a common namespace within the framework. The registry model used in each is that of nested collections of attribute-value pairs, where the nested collections form a hierarchy. Any change to a registry generates an event that agents may receive in order to perform subsequent activities. The registry of the release dock mostly provides a list of available software releases, whereas the registry of the field dock performs the valuable role of providing access to consumer-side information.

Consumer-side information is critical in performing nearly any software deployment pro-

cess. In the past, software deployment was complicated by the fact that consumer-side information was not available in any standardized fashion. The field dock registry addresses this issue by creating a detailed, standardized, hierarchical schema for describing the state of a particular consumer site. By standardizing the information available within a consumer organization, the field dock creates a common software deployment namespace for accessing consumer-side properties, such as operating system and computing platform. This information, when combined with the description of a software system, is used to perform specific software deployment processes.

Agents implement the actual software deployment process functionality. When the installation of a software system is requested on a given consumer site, initially only an agent responsible for installing the specific software system and the description of the specific software system are loaded onto the consumer site from the originating release dock. The installation agent docks at the local field dock and uses the description of the software system and the consumer site state information provided by the field dock to configure the selected software system. When the agent has configured the software system for the specific target consumer site, it requests from its release dock the precise set of artifacts that correspond to the software system configuration.

The installation agent may request other agents from its release dock to come and dock at the local field dock. These other agents are responsible for other deployment activities such as update, adapt, reconfigure, and remove. Each agent performs its associated process by interpreting the information of the software system description and the consumer site configuration.

The wide-area event service in the Software Dock architecture provides a means of connectivity between software producers and consumers for "push"-style capabilities. Agents that are docked at remote field docks can subscribe for events from release docks and can then perform actions in response to those events, such as performing an update. Siena (Section 3.1.5) is currently used for event notification in the Software Dock. In addition to event notification, direct communication between agents and release docks is supported and provided by standard protocols over the Internet. Both forms of connectivity (events and direct messages) combine to provide the software producer and consumer the opportunity to cooperate in their pursuit of software deployment process support.

Deployable Software Description. In order to automate or simplify software deployment processes it is necessary to have some form of deployment knowledge about the software system being deployed. One approach to this requirement is the use of a standardized language or schema for describing a software system; this is the approach adopted by the Software Dock research project. In such a language or schema approach it is common to model software systems as collections of properties, where semantic information is mapped into standardized properties and values.

The Software Dock project has defined the Deployable Software Description (DSD) format

to represent its system knowledge. The DSD is a critical piece of the Software Dock research project that enables the creation of generic deployment process definitions. The DSD provides a standard schema for describing a software system *family*. In this usage, a family is defined as all revisions and variants of a specific software system. The software system family was chosen as the unit of description, rather than a single revision, variant, or some combination, because it provides flexibility when specifying dependencies, enables description reuse, and provides characteristics, such as extending revision lifetime, that are necessary in component-based development.

We have identified five classes of semantic information that must be described by the software system model. These classes of semantic information are:

- Configuration - describes relationships inherent in the software system, such as revisions and variants, and describes resources provided by the software system, such as deployment-related interfaces and services.
- Assertions - describe constraints on consumer-side properties that must be true otherwise the specific deployment process fails, such as supported hardware platforms or operating systems.
- Dependencies - describe constraints on consumer-side properties where a resolution is possible if the constraint is not true, such as installing dependent subsystems or reconfiguring operating system parameters.
- Artifacts - describe the actual physical artifacts that comprise the software system.
- Activities - describe any specialized activities that are outside of the purview of standard software deployment processes.

A DSD family description is broken into multiple elements that address the five semantic classes of information. The sections of a DSD family description are identification, imported properties, system properties, property composition, assertions, dependencies, artifacts, interfaces, notifications, services, and activities. Some of these sections map directly onto the five semantic classes of information, others, such as system properties, property composition, interfaces, and notifications, combine to map onto the configuration class of semantic information. For more information about the DSD, refer to publications [13], [14], [15], and [27] in Section 5.

Enterprise Software Deployment. Enterprise software deployment extends the current single site software deployment to the problem of managing the integrity of software systems on many sites throughout an organization. This extension requires that enterprise software deployment deal with issues of scale, distribution, coordination, and heterogeneity. The low-level details of the various software deployment life cycle processes are therefore not the focus of

enterprise software deployment; the focus is coordinating and managing deployment processes across multiple sites.

For example, installing a software system on a thousand sites reveals issues that are not present when installing the same software system on a single site. Complications arise due to the necessity to consider policy decisions, such as ad hoc, phased-in, or all-or-nothing installation. Also, heterogeneity issues are very important when dealing with a large number of sites since the software deployment processes depend heavily on the precise configuration of a site's hardware, operating system, and resources.

In order to provide a solution for enterprise software deployment, it is necessary for a symbiotic relationship to exist between standard software deployment and enterprise software deployment. Enterprise software deployment must build on top of a standard software deployment solution. The current Software Dock prototype provides limited support for enterprise level operation (see the Admin Workbench discussion below); it remains an ongoing research topic.

Prototype. The current Software Dock implementation includes a field dock, a release dock, and a collection of generic agents for performing the install, update, adapt, and removal of DSD described software systems. Additional tools, such as the Schema Editor for creating DSD descriptions and the Docking Station for managing software at a field dock, are also provided. The Software Dock is implemented entirely in Java and uses the remote procedure call and agent capabilities of ObjectSpace's Voyager, which is also completely Java-based.

The prototype of the Software Dock provides the primary field dock interface shown in Figure 12. From this interface, a user at the field dock can carry out various life cycle activities including install of a new system and update, reconfigure, adapt, or remove of a previously installed system. Most of these activities involve specifying various properties of the system. Figure 13 shows the interface to the generic mechanism for defining or modify the properties associated with a system.

Enterprise level operations are represented by the Admin(istrator) Workbench shown in Figure 14. The Admin Workbench provides an entry point for software administrators to monitor the result of deployment activities on managed sites, as well as, to perform remote operations such as taking an inventory or pushing updates or reconfigurations.

This interface is still experimental since the set of enterprise-level operations is still in flux. This current interface allows an administrator to do a variety of things.

- Monitor the activities of field docks,
- Take inventory of the systems installed at one or more field docks,
- Force reconfigurations, removals, updates, and adaptations upon one or more field docks.
- Enforce constraints on allowable configurations upon field docks.

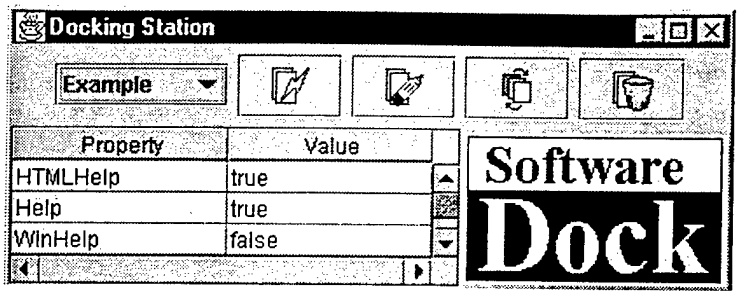

Figure 12: Field Dock Main Interface.

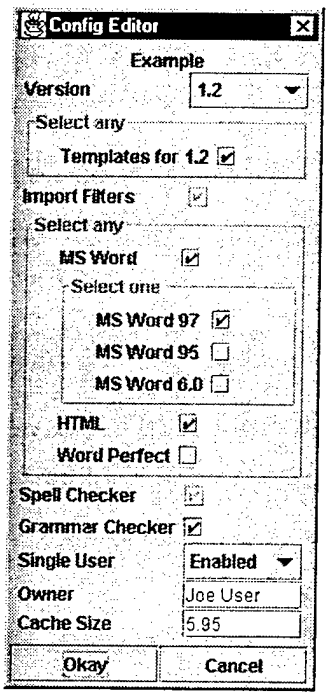

Figure 13: Field Dock Property Manipulation Interface.

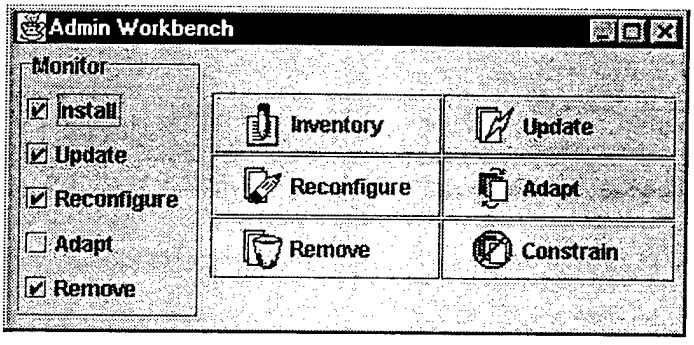

Figure 14: Enterprise-level Administrators Workbench Interface.

	Software Dock	InstallShield
Install	172.0s	168.0s
Remove	36.7s	80.0s
Reconfig (remove)	40.3s	90.0s
Reconfig (add)	113.3s	284.3s
Update	187.3s	149.6s

Table 1: Software Dock Performance Comparison.

Note that the communication between the administrator and the field docks and release docks is provided by Siena (Section 3.1.5).

Experience. The current implementation was used in two joint demonstrations with Lockheed Martin Corporation at several of the EDCS "Demo Days" activities.

The first demonstration used a Web-based software system called the Online Learning Academy (OLLA), which consisted of 45 megabytes of data and software in over 1700 files. OLLA was comprised of two dependent subsystems called Disco and Harvest. The software deployment processes of release, install, reconfigure, update, adapt, and remove were all initially demonstrated using the generic agents along with the DSD description of all three software systems.

The Second demonstration involved the use of the Software Dock with the Lockheed EVOLVER project and was demonstrated at the Baltimore Demo Days meeting. A core technology of EVOLVER was a KQML-based mechanism for wrapping information sources and making them available through the EVOLVER infrastructure. This involved two steps. First, an information source was made available in a simple form by providing a KQML wrapper. The second step require that the wrapped information source also export meta-information that allowed EVOLVER to infer connections between the information source and other sources available through EVOLVER.

We obtained an early release of the Java-based KQML wrapper system from Lockheed Martin, and we applied it to the Software Dock to make the Dock's repository of configuration information available through EVOLVER. Although there were some problems, the integration was successfully completed. The biggest hurdle was to map between the Software Dock's data model and the EVOLVER data model.

Experiments were also conducted to verify the performance of the Software Dock. These experiments compared the Software Dock prototype to an existing deployment solution (i.e., InstallShield) for a specific software system. A DSD specification for versions 1.1.6 and 1.1.7 of the Java Development Kit (JDK) by Sun Microsystems was created in order to compare the Software Dock deployment processes to the standard InstallShield self-extracting distribution archive for the Microsoft Windows platform. Time to completion was the dimension for comparison. Table 1 summarizes the results of the experiments.

The Software Dock performed as well or better than InstallShield in most cases, despite the

fact that file artifacts were dynamically packaged for the specific configuration requests. This dynamicity was most obvious in the update process. The comparison is strained in the case of update and reconfigure because standard InstallShield package for JDK does not properly perform these activities, and it does not perform adapts at all.

3.1.5 Siena

There is a clear trend among experienced software developers toward designing large-scale distributed systems as assemblies of loosely-coupled autonomous components; a trend that was evident in EDCS especially. One approach to achieving loose coupling is an *event-based* or *implicit invocation* design style. In an event-based system, component interactions are modeled as asynchronous occurrences of, and responses to, *events*. To inform other components about the occurrences of internal events (such as state changes), components emit *notifications* containing information about the events. Upon receiving notifications, other components can react by performing actions that, in turn, may result in the occurrence of other events and the generation of additional notifications.

Several classes of applications make use of some sort of event service. Examples of such applications are monitoring systems, user interfaces, integrated software development environments, active databases, software deployment systems, content distribution, and financial market analysis. Many of these applications are also inherently distributed, and thus they require interaction among components running on different sites and possibly distributed over a wide-area network.

Wide-area networks such as the Internet, with their vast number of potential producers and consumers of notifications, create an opportunity for developing novel distributed event-based applications in such fields as market analysis, data mining, indexing, and security. In general, the asynchrony, heterogeneity, and inherent high degree of loose coupling that characterize applications for wide-area networks suggest event interaction as a natural design abstraction for a growing class of distributed systems. Yet to date there has been a lack of sufficiently powerful and scalable middleware infrastructures to support event-based interaction in a wide-area network. We refer to such a middleware infrastructure as an *event notification service*.

Siena [2, 3, 4, 9, 18] is our prototype Internet-scale event notification service that is representative of the capabilities we envision for scalable event notification middleware. Siena is designed to be a ubiquitous service accessible from every site on a wide-area network.

Architecture As shown in Figure 15, Siena is implemented as a distributed network of servers that provide clients with *access points* offering an extended publish/subscribe interface. The clients are of two kinds: *objects of interest*, which are the generators of notifications, and *interested parties*, which are the consumers of notifications; of course, a client can act as both an object of interest and an interested party. Clients use the access points of their local servers to *publish* their notifications. Clients also use the access points to *subscribe* for individual

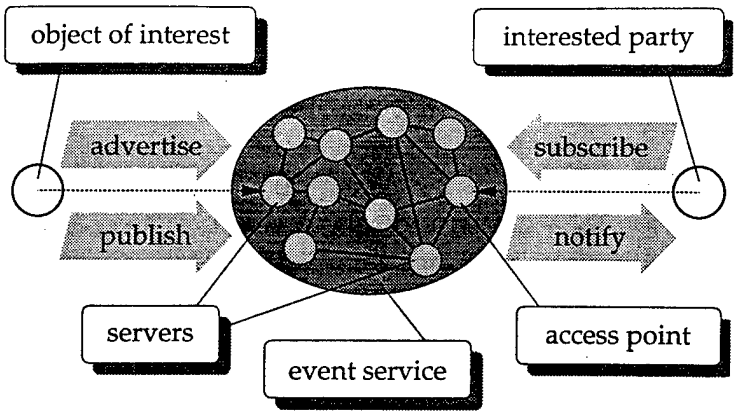

Figure 15: Distributed Event Notification Service.

notifications or compound patterns of notifications of interest. Siena is responsible for *selecting* the notifications that are of interest to clients and then *delivering* those notifications to the clients via the access points.

Siena is a *best-effort* service in that it does not attempt to prevent race conditions induced by network latency. This is a pragmatic concession to the realities of Internet-scale services, but it means that clients of Siena must be resilient to such race conditions. For instance, clients must allow for the possibility of receiving a notification for a cancelled subscription. Of course, an implementation would likely adopt techniques such as persistent data structures, transactional updates to the data structures, and reliable communication protocols to enhance the robustness of this best-effort service.

The key design challenge faced by Siena is maximizing *expressiveness* in the selection mechanism without sacrificing *scalability* of the delivery mechanism. The scalability problem can be characterized by the following dimensions:

- large number of objects publishing events and subscribing for notifications,
- large number of events,
- high event generation rates,
- objects distributed over a wide-area network (thus, low bandwidth, scarce connectivity and reliability),
- events of the same class generated by many different objects,
- notifications of the same class of events requested by many objects,
- no centralized control nor global view of the structure of the event service.

Expressiveness refers to the power of the data model that is offered to publishers and subscribers of notifications. Clearly the level of expressiveness influences the algorithms used to route and deliver notifications, and the extent to which those algorithms can be optimized. As the power of the data model increases, so does the complexity of the algorithms. Therefore, the expressiveness of the data model ultimately influences the scalability of the implementation, and hence scalability and expressiveness are two conflicting goals that must be traded off.

While we have not fully explored the nature of this tradeoff, we have investigated a number of carefully chosen points in the tradeoff space. In particular, we designed a data model for Siena that we believe is sufficiently expressive for a wide range of applications while still allowing sufficient scalability of the delivery mechanism. Based on this data model, we designed distributed server architectures and associated delivery algorithms and processing strategies, and we evaluated and confirmed their scalability.

Interface The interface of the Siena event service allows objects to subscribe for specific classes of events, by setting up filters, or for specific sequences of events, by setting up patterns. Filters select events based on their content using a simple and yet powerful query language. Patterns are combinations of filters that select temporal sequences of events.

Siena provides a flexible notification model that can serve application programmers as well as end users. Event notifications (Figure 16) are structured as a set of attributes. Each attribute has a name, a type, and a value.

Event filters (Figure 17) are structured as a set of simple relations. Each relation is an attribute name, an operator, and a constant value.

Routing Optimization Siena delivers a scalable event service by adopting special dispatching protocols that aim at reducing network traffic and avoiding bottlenecks.

Depending on the topology of connections among Siena servers, hierarchical or peer-to-peer, different algorithms have been implemented to deliver notifications. These are based on the propagation of subscriptions (subscription forwarding) or on the propagation of advertisements (advertisements forwarding). These two algorithms also roughly correspond to the main strategies that Siena applies in filtering and multicasting notifications:

- upstream filtering and assembly: filters and patterns are pushed as close as possible to the sources of events, thereby immediately pruning the propagation of notifications that are not requested by any object.

- downstream replication: replication of notifications (multicasting) is pulled as close as possible to the targets of notifications. The idea being that, in order for a notification to reach several objects on distant networks, only one copy of that notification needs to traverse slow internetwork paths. That notification is then replicated and routed to all its destinations only when it gets to their local (less expensive) network.

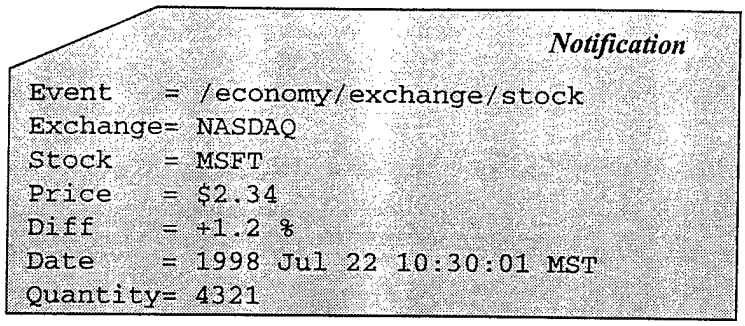

Figure 16: Siena Event Notification Example.

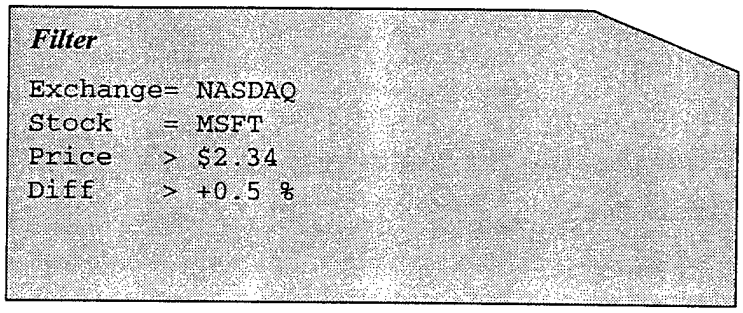

Figure 17: Siena Event Filter Example.

Figure 18 illustrates an example of hierarchical routing in Siena.

Experience. The design of Internet-scale systems requires a special effort for validation. In particular, it is important to assess the impact of routing strategies and event pattern recognition with respect to costs such as network traffic, CPU, and memory usage.

The architectures of Siena and its routing algorithms were studied by means of systematic simulations in various network scenarios with different ranges of loads and different configurations.

Currently, a prototype of Siena is used as wide-area messaging and event system of the Software Dock. There are two main implementations of the Siena server. One (written in Java) realizes a hierarchical server, while the other (written in C++) has a peer-to-peer architecture. The client interface is currently available for both Java and C++.

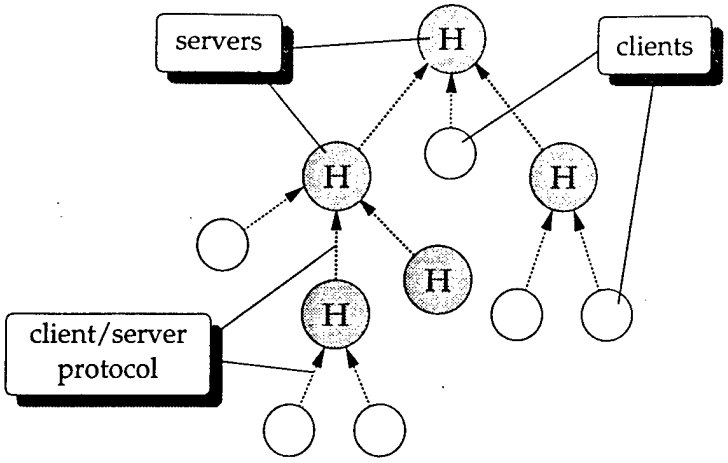

Figure 18: Hierarchical Routing Example.

3.2 Technical Transfer

3.2.1 Prototype Availability

The University of Colorado EDCS project prototypes are generally available via the World Wide Web. Potential users are especially encouraged to obtain this software via SRM (*http://www.cs.colorado.edu/serl/software*). Using SRM will ensure that the user will obtain, in a single step and in a single package, all the necessary components required to install any of our software systems. A detailed descriptions of each product, as well as the software itself, is available at that page. In addition, users are encouraged to visit the University of Colorado SERL web site (*http://www.cs.colorado.edu/serl*) to obtain background material related to these prototypes and to obtain copies of the publications described in Section 5.

3.2.2 Other Technical Transfer Efforts

Recently, we were contacted by Harriet Cohen, Director of Product Management for Content-Integrity, Inc. They are a Boston-based start-up that is in the final stages of beta test with a product that embodies key configuration management concepts developed under this contract.

Dassault Systems explored the use of the Software Dock style architecture and approach to support the electronic deployment of their software systems. Their first target is the Dassault CATIA system, which is a very large CAD-CAM system consisting of a core system plus some 150 independently deployable application sub-systems. CATIA is used by numerous large companies including Boeing and Chrysler, and the first deployment experiments will be joint with Boeing.

The Software Dock has been provided to Nortel Networks to support their experiments in push deployment of telecommunications software. As mentioned in Section 3.1.4, we also integrated the Software Dock with two demonstration systems from Lockheed Martin.

The University of Massachusetts LASER project has adopted NUCM/SRM as their standard software release mechanism.

The software developed under this project has been released using SRM under the University of Colorado Software Engineering Research Laboratory (SERL) web site. SERL software has been downloaded over 600 times. This includes SRM, NUCM, and DVS. Based on these downloads, many organizations have registered as interested parties for this software so that they can be notified of updates and changes to the software.

SRM was used as the primary release mechanism for software produced by the EDCS program. A central server, located at the Software Engineering Institute, served as the repository to which participating organizations released their systems. Subsequently, these systems were retrieved by users from all over the world.

3.3 Students

The education and graduation of students is, of course, a primary activity for a Research University such as the University of Colorado. This project has wholly or partially supported a number of outstanding graduate students. Table 2 lists them alphabetically by last name.

Student	Degree and Date	Dissertation Title	Current Employment
Antonio Carzaniga	Ph. D. 1999	Architectures for an Event Notification Service Scalable to Wide-area Networks	Research Associate, University of Colorado
John C. Doppke	M. S. 1998	Software Process Modeling and Execution Within Virtual Environments	Consultant
Richard Hall	Ph. D. 1999	Agent-based Software Configuration and Deployment	Asst. Professor, Free University of Berlin
André van der Hoek	Ph. D. 2000	A Reusable, Distributed Repository for Configuration Management Policy Programming	Asst. Professor at the University of California, Irvine
Judith Stafford	Ph. D. 2000	A Formal, Language-Independent, and Compositional Approach to Control Dependence Analysis	MTS at the Software Engineering Institute
Carlton Reid Turner	Ph. D. 1998	Feature Engineering of Software Systems	Lincap Corporation

Table 2: Alphabetical List of Graduated Students Associated with this Contract.

4 Conclusions

The University of Colorado EDCS project has been successful in achieving its objective: producing innovative, useful, and interesting research results in the areas of software configuration and deployment.

These research results were embodied in five prototype systems developed in whole or part under this project.

1. *NUCM* – a generic, tailorable, peer-to-peer repository supporting distributed Configuration Management.

2. *SRM* – a tool to manage the release of multiple, interdependent software systems from distributed sites.

3. *DVS* – a tool to support distributed authoring and versioning of documents with complex structure, and to support multiple developers at multiple sites over a wide-area network.

4. *Software Dock* - a distributed, agent-based framework supporting software system deployment over a wide-area network.

5. *Siena* - an Internet-scale distributed event notification service allowing applications and people to coordinate in such activities as updating software system deployments.

The results from this project have been widely disseminated in the form of reports, articles, and other publications; software distributions to over 600 sites; technical transfers to commercial practice; and graduating quality Ph. D. and M. S. students.

5 References and Bibliography

Reverse Chronological List of Publications Funded by this Grant.

[1] J.A. Stafford and A.L. Wolf, "Annotating Components to Support Component-Based Static Analyses of Software Systems," Proceedings of Grace Hopper Conference 2000, September 2000, Hyannis, MASS.

[2] A. Carzaniga, D.S. Rosenblum, and A.L.Wolf, "Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service," Proc. of the 19th ACM Symposium on Principles of Distributed Computing, July 2000, Portland OR.

[3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Content-Based Addressing and Routing: A General Model and its Application," Technical Report CU-CS-902-00, January 2000, Department of Computer Science, University of Colorado, Boulder.

[4] A. Carzaniga, D.S. Rosenblum, and A.L.Wolf, "Challenges for Distributed Event Services: Scalability vs. Expressiveness," ICSE 99 Workshop on Engineering Distributed Objects (EDO'99), May 1999, Los Angeles CA.

[5] A. van der Hoek, "A Reusable, Distributed Repository for Configuration Management Policy Programming," Ph. D. Thesis, January 21, 2000, Department of Computer Science, University of Colorado, Boulder.

[6] C. Reid Turner, A. Fuggetta, L. Lavazza, and A.L. Wolf, "A Conceptual Basis for Feature Engineering," Journal of Systems and Software, 49(1):3–15 (December 1999).

[7] A. van der Hoek, D. Heimbigner, and A.L. Wolf, "Capturing Architectural Configurability: Variants, Options, and Evolution," Technical Report CU-CS-895-99, December 1999, Department of Computer Science, University of Colorado, Boulder.

[8] J.A. Stafford and A.L. Wolf, "Annotating Components to Support Component-Based Static Analyses of Software Systems," Technical Report CU-CS-896-99, December 1999, Department of Computer Science, University of Colorado, Boulder.

[9] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Interfaces and Algorithms for a Wide-Area Event Notification Service," Technical Report CU-CS-888-99, October, 1999, Department of Computer Science, University of Colorado, Boulder.

[10] Andre van der Hoek, "Configurable Software Architecture in Support of Configuration Management and Software Deployment," Proc. of the Doctoral Workshop of the 1999 Int'l. Conf. on Software Engineering, pages 732 - 733. May 1999, Los Angeles, CA.

[11] Richard Hall, Dennis Heimbigner, and Alexander L. Wolf, "A Cooperative Approach to Support Software Deployment Using the Software Dock," Proc. of ICSE'99: The 1999 Int'l Conf. on Software Engineering, pages 174-183, May 1999, Los Angeles, CA.

[12] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf, "An Architecture-Based Approach to Self-Adaptive Software," IEEE Intelligent Systems Special Issue on Self-Adaptive Software, 14(3):54-62 (May/June 1999).

[13] D. Heimbigner, R.S. Hall, and A.L. Wolf, "A Framework for Analyzing Configurations of Deployable Software Systems," Proc. of the Fifth IEEE Int'l Conference on Engineering of Complex Computer Systems, pp. 32–42, October 1999, Las Vegas, NV.

[14] Richard Hall, "Agent-based Software Configuration and Deployment," Ph. D. Thesis, April 1, 1999, Department of Computer Science, University of Colorado, Boulder.

[15] R.S. Hall, D. Heimbigner, and A.L. Wolf, "Specifying the Deployable Software Description Format in XML," Technical Report CU-SERL-207-99, March 1999, Software Engineering Research Laboratory, Department of Computer Science, University of Colorado, Boulder.

[16] Daniele Compare, Paola Inverardi, and Alexander L. Wolf, "Uncovering Architectural Mismatch in Component Behavior," Science of Computer Programming, 33(2) (Feb. 1999).

[17] Carlton Reid Turner, "Feature Engineering of Software Systems," Ph. D. Thesis, Dec. 1998, Department of Computer Science, University of Colorado, Boulder.

[18] Antonio Carzaniga, "Architectures for an Event Notification Service Scalable to Wide-area Networks," Dec. 1998, Ph. D. Thesis, Politecnico di Milano,

[19] A. Carzaniga, E. Di Nitto, D.S. Rosenblum, and A.L. Wolf, "Issues in Supporting Event-Based Architectural Styles," 3rd International Software Architecture Workshop (ISAW3), November, 1998, Orlando, FL.

[20] Judith A. Stafford and Alexander L. Wolf, "Architecture-Level Dependence Analysis in Support of Software Maintenance," Proc. of the 3rd Int'l Software Architecture Workshop, November 1998, Orlando, FL.

[21] Andre van der Hoek, Dennis Heimbigner, and Alexander L. Wolf, "Versioned Software Architecture," Proc. of the 3rd Int'l Software Architecture Workshop, November 1998, Orlando, FLA.

[22] Richard Hall, Dennis Heimbigner, and Alexander L. Wolf, "Evaluating Software Deployment Languages and Schema: An Experience Report," Proc. of the 1998 Int'l Conf. on Software Maintenance, November 1998, Bethesda, MD,

[23] J.E. Cook and A.L. Wolf, "Event-Based Detection of Concurrency," Proc. of the 6th Int'l Symposium on Foundations of Software Engineering, pages 35-45, November 1998, Orlando, FLA.

[24] Judith A. Stafford and Alexander L. Wolf, "'Dependence Analysis for Software Architectures," Proc. of the ASE'98 Doctoral Symposium, October 1998, Honolulu, HI.

[25] Andre van der Hoek, Dennis Heimbigner, and Alexander L. Wolf, "Investigating the Applicability of Architecture Description in Configuration Management and Software Deployment," Technical Report CU-CS-862-98, September 1998, Department of Computer Science, University of Colorado, Boulder.

[26] Andre van der Hoek, Antonio Carzaniga, Dennis Heimbigner, and Alexander L. Wolf, "A Generic, Reusable Repository for Configuration Management Policy Programming," Technical Report CU-CS-864-98, September 1998, Department of Computer Science, University of Colorado, Boulder.

[27] Richard Hall, Dennis Heimbigner, and Alexander L. Wolf, "Requirements for Software Deployment Languages," Proc. of the 8th Int'l Software Configuration Management Workshop, July 1998, Brussels, Belgium.

[28] Andre van der Hoek, Dennis Heimbigner, and Alexander L. Wolf, "System Modeling Resurrected," Proc. of the 8th Int'l Software Configuration Management Workshop, July 1998, Brussels, Belgium.

[29] Judith A. Stafford, "Aladdin: A Tool for Analysis of Dependencies in Software Architectures," Presentation for the Annual Symposium on Software Engineering and Technology Transfer (ASSETT 1998), July 16, 1998, Motorola Museum, Schaumburg, IL.

[30] J.E. Cook and A.L. Wolf, "Balboa: A Framework for Event-Based Process Data Analysis," Proc. Fifth Int'l Conf. on the Software Process, pp. 99–110, June 1998, Lisle, IL.

[31] Judith A. Stafford, Debra J. Richardson, and Alexander L. Wolf, "Architecture-level Dependence Analysis for Software Systems," Proc. of the Int'l Workshop on the Role of Software Architecture in Testing and Analysis, June 1998, Marsala, Italy.

[32] A. Carzaniga, A. Fuggetta, R. S. Hall, A. van der Hoek, D. Heimbigner, and A. L. Wolf, "A Characterization Framework for Software Deployment Technologies,"

Technical Report CU-CS-857-98, April 98, Department of Computer Science, University of Colorado, Boulder.

[33] A. van der Hoek, R.S. Hall, A. Carzaniga, D. Heimbigner, and A.L. Wolf, "Software Deployment: Extending Configuration Management Support into the Field," *Crosstalk, The Journal of Defense Software Engineering*, 11(2) (February 1998).

6 Symbols, Abbreviations, and Acronyms

BADD	Battlefield Awareness
CM	Configuration Management
CPOF	Command Post of the Future
DSD	Deployable Software Description
DVS	Distributed Versioning System
EDCS	Evolutionary Design of Complex Software
GIG	Global Information Grid
JBI	Joint Battlespace Infosphere
NUCM	Network Unified Configuration Management
RCS	Revision Control System
SERL	Software Engineering Research Laboratory (at the University of Colorado)
Siena	Scalable Internet Event Notification Architectures
SRM	Software Release Manger
WebDAV	Web-based Distributed Authoring and Versioning
XML	Extensible Markup Language